FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Programming Shared Virtual Memory Multiprocessors

Michael Gerndt

KFA-ZAM-IB-9526

November 1995
(Stand 22.11.95)

Keynote paper to be published in
Proceedings of Euromicro's 4th Workshop on Parallel and Distributed
Processing 96, January 24-26, 1996, in Braga, Portugal (Copyright by IEEE)

Programming Shared Virtual Memory Multiprocessors

Michael Gerndt
Central Institute for Applied Mathematics
Research Centre Jülich (KFA)
D-52425 Jülich, Germany
email: m.gerndt@kfa-juelich.de

Abstract

Highly parallel machines needed to solve compute intensive scientific applications are based on the distribution of physical memory across the compute nodes. The drawback of such systems is the difficult message passing programming model. Therefore, a lot of research in simplifying the programming model is going on. This article investigates the combination of a task parallel programming model implemented on top of a shared virtual address space provided by the operating system of the parallel machine.

1 Programming Models for Massively Parallel Systems

A large number of numerical simulations require highly parallel computing systems for reasonable turnaround times. Current architectures are based on the distribution of memory across the compute nodes to provide the necessary memory bandwidth. The scalability of centralized shared memory systems is limited by the network and the memory bandwidth.

Programming languages and tools for distributed memory systems and thus for the underlying programming models hide to a different extent the two important properties of such systems: parallelism and memory distribution.

Currently three programming models are available for distributed memory multiprocessors: message passing, data parallel, and task parallel.

All models are based on a static process concept: A set of processes is created at the beginning of the application and run until its end. In the message passing model, the user has to provide local code for each process and a process can access only private data. Access to remote data has to be programmed in form of message passing.

In the data parallel programming model, the program is written in a global style with a single thread of control and parallel operations on data structures. Data are accessed in parallel operations via simple variable references. The distribution of data structures determines the distribution of parallel computations.

In the task parallel programming model, the program is also written in a global style. Tasks, for example parallel sections or iterations of parallel loops, are distributed onto the processes. Processes can access shared and private data via variable references.

All three programming models force the user to specify parallelism. Automatic parallelization tools can be used to detect parallelism in sequential codes and express it in the data parallel or the task parallel programming model.

Due to the shared address space in the data parallel and the task parallel programming model, and the support of global parallel code, i.e. vector operations or parallel loops, coding parallel algorithms is much easier than in the message passing model. Of course, in both models the user has to be aware of the physical distribution of memory and has to ensure that computations are done by the processor which has the accessed data in its local memory already.

Mapping computation and data in an optimal way has to be performed manually by the user. The two models provide different concepts for this task: the data distribution concept and the work distribution concept. While data distributions determine explicitly the data layout across the physical memories and implicitly the work distribution via the owner-computes rule, the work distribution is explicitly specified in the task parallel model. The programmer simply has to map computations accessing the same data[1] to the same processor to ensure data locality instead of worrying about the individual distributions of all the data structures of a program.

The most well-known programming language based

[1] Typically, these are computations for the same object in the physical domain of the application, and the work distribution directly results from the domain decomposition selected for parallelization.

on the data parallel model is HPF [HPF 94]. It was designed for *distributed private memory multiprocessors* which do not support a global address space on system level. The HPF compiler has to generate message passing code, i.e. the program is translated into local code executed on each processor and remote references have to be translated into explicit message passing. Both tasks require precise knowledge of the distributions of data structures and an analysis of the access patterns. This requirement lead to a very restrictive design of the language.

With the advent of *distributed shared memory systems (DSM systems)*, the implementation of HPF was simplified. DSM systems support a global address space, either in hardware or in software. Distributed arrays can be mapped into the global address space and no message passing is required to access remote data. But still the compiler has to understand the distribution of data structures to determine the distribution of parallel operations.

Due to the global address space in DSM systems, on such machines the much more flexible task parallel model can be implemented. The work distribution specification in this model eliminates the second implementation problem mentioned above. As stated previously, the model is based on the dynamic exploitation of local memory if data is reused in a processor. This is offered in DSM systems either by big caches in the individual nodes, we call these systems *cache-coherent remote memory access systems (CC-RMA systems)* or by migrating the data on demand into the local memory, as it is done in *shared virtual memory systems (SVM systems)*.

2 Shared Virtual Memory Systems

In CC-RMA systems like the Convex Exemplar, DASH and Flash [LLGW 92], each address has a fixed home memory. Remote addresses can be cached and thus reuse of a remote address in a processor may result in a local access. If a remote address is evicted from the cache, it is written back to its fixed home memory. The concept of caching has at least two important aspects: First, the coherence of copies in the caches of multiple processors has to be guaranteed, and second, the cache has to be large enough to avoid premature eviction of data. The second aspect is particularly important for scientific codes with their big arrays. The cache coherence problem has been studied in a lot of research projects, and several different techniques have been developed. Recently, a cache coherence protocol was standardized in the IEEE SCI-Bus [JLGS 90].

In SVM systems, a virtual address does not have a fixed home memory. If a currently remote address is accessed, a block of memory[2] including this address is transfered into the local memory of the node. Therefore, the next accesses to this address range in the processor are serviced with high probability by the local memory. In addition, an address that is evicted from the processor cache due to capacity problems is always written back to the local memory.

Most SVM systems are implemented in software based on the memory management hardware of the processor. The systems from Kendall Square were the only hardware implementation of SVM. In this article we focus on software SVM systems that can be implemented on top of every distributed private memory architecture.

Similar to CC-RMA systems, SVM systems have to solve the coherence problem for multiple copies of pages in different memories. Due to the lack of a fixed home location, SVM systems have to manage the dynamically changing virtual address mapping of pages.

The basis for all SVM implementations is the dissertation of Kai Li [Li 86]. His implementation called IVY demonstrated the feasibility of this concept [Li 88]. He described the implementation of strong coherence via an invalidation approach[3] and introduced several manager algorithms for the administration of pages.

The manager algorithms vary in their network requirements and the load balance of the overhead. In the *broadcast distributed manager algorithm*, a request for a page is broadcasted to all processors and served by the processor responsible for the page. A single processor keeps all the information and services all the requests in the *centralized manager algorithm*. In the *fixed distributed manager algorithm* the responsibility for pages is cyclically distributed among processors. The approach taken by most SVM implementations is the *dynamic distributed manager algorithm*, which distributes the responsibility dynamically according to the access patterns and thus leads to optimal results for static communication patterns.

Besides network contention and load balance, the memory requirements of the algorithms are very important. This aspect was studied in a recent SVM-implementation for the Intel Paragon, the *Advanced Shared Virtual Memory system (ASVM)* [ZTM 96, Zeis 93]. In the dynamic distributed manager algorithm each processor has to maintain for each page in the global address space some administration infor-

[2] Typically, this is done on page or cache line basis.

[3] This protocols guarantees coherence by allowing multiple read copies but only one copy if a processor writes to a page

mation, which limits the number of nodes in an SVM system. In the ASVM the dynamic distributed manager algorithm is based on a cache for administration information to reduce its memory requirements. If no information is available in the cache, other protocols are applied to obtain the required information.

The strong coherent memory based on the big granularity of pages stresses a problem also known from CC-RMA systems, *false sharing*. Although processors access different locations on a page, the page is migrated between the processors. One way to reduce this problem are weaker coherence protocols also known from CC-RMA systems. The first SVM system supporting a weak coherence protocol is the *KOAN* implementation [BoPr 92] on the iPSC/2. Pages in a weak coherent address range may have multiple write copies. Only when the address range is explicitly made coherent, the write copies are merged to form again a single copy of the page.

A new approach in implementing SVM is *BLIZZARD-E* on the CM-5 [SFL 94]. In addition to the page-based techniques described above, BLIZZARD-E uses the *error correcting code bits (ECC bits)* of the memory to implement much finer-grain coherence units. The ECC bits are modified such that an exception occurs if an address is accessed which does not have appropriate permissions. In addition to the new functionality, the ECC bits still fulfill in most points their original purpose.

3 Programming Languages Supporting the Task Parallel Model

Task parallel programming languages are more general than data parallel programming languages. With respect to scientific applications they offer functional parallelism for implementing multi-disciplinary applications and more flexible work distribution features. As mentioned above, the important feature with respect to SVM systems is the ability to specify the work distribution.

First language constructs for work distribution were implemented in KSR Fortran [KSRF 92]. Parallel loops could be annotated with regular distribution strategies, such as block and cyclic. In addition, the affinity region concept was provided with the following notation:

```
C*KSR* affinity region ( parameter list )
  code including multiple parallel loops
C*KSR* end affinity region
```

The parameter list allows the programmer to specify the distribution of an iteration space onto the processors. The iteration spaces of parallel loops within the affinity region are distributed onto the processors according to that specification. The affinity region has the very interesting property of defining a work distribution in a global way for a specified code region. It neither supports the usage of multiple work distributions in the same code region nor global work distribution across subroutine boundaries, and does not support work distribution for functional parallelism.

Another drawback of KSR Fortran was the underlying fork-join model. In the fork-join model new processes are dynamically created for parallel regions and thus scheduling work onto processes does not guarantee locality. Fortran-S [BKP 93] instead was based on a static set of processes created at the beginning of the application and terminated at its end. It is the programming interface for the KOAN implementation on the iPSC/2 and the successor MYOAN on the Intel Paragon. Fortran-S provides similar work distribution concepts as KSR Fortran but, due to the static process concept, the resulting mapping is much more intuitive to the user.

The limitations of the work distribution concept in both languages triggered the design of a new language called SVM-Fortran [BeGe 95]. SVM-Fortran is one component of the SVM-Fortran programming environment which is based on the ASVM on the Intel Paragon and additionally provides performance analysis and optimization tools[4].

SVM-Fortran borrowed the concepts of processor arrangements and templates from HPF as tools to specify scheduling decisions globally through template distributions. In contrast to the affinity region of KSR, individual templates can be used for different objects in the application domain and scheduling strategies can be defined for the whole program.

Due to the implementation requirements, the template concept in HPF is very restrictive. SVM-Fortran, targeting SVM systems, uses templates in a much more flexible way. Templates can not only be distributed with simple standard strategies, i.e. block and cyclic, but the general-block strategy known from Vienna Fortran and an element-wise irregular distribution can be used. The irregular distribution is very useful for unstructured grid applications.

In addition, templates do not have static distributions by default. Since the distributions are evaluated at runtime, the compiler does not have to know the distribution. Therefore, templates can be redistributed at any time during the computation. Due

[4]More information about the SVM-Fortran project can be accessed via the world wide web: http://www.kfa-juelich.de/zam/PT/ReDec/ProgLangc/SVM.html

```
      SUBROUTINE G(....,T,N,...)
CSVM$ TEMPLATE::  T1(N,N)
CSVM$ PROCESSORS::  P(2,NUMPROC()/2)
CSVM$ DISTRIBUTE (BLOCK,BLOCK) ONTO P::T1

CSVM$ PDO(LOOPS(I,J),
CSVM$*   STRATEGY(ON_HOME(T1(I,J))))
      DO I=1,M
         DO J=1,M
            ...
         ENDDO
      ENDDO
```

Figure 1: SVM-Fortran example program

to the same reasons, templates can be passed to subroutines as arguments and thus, a distribution can be defined globally in the main program and need not be specified in each subroutine separately.

Template distributions determine the work distribution for parallel loops in *predefined scheduling* which assigns loop iterations to processors according to the distribution of the template.

In addition to scheduling loops based on a template's distribution, the template's distribution can also result from scheduling a loop. This is especially useful for codes with unbalanced work load and with irregular accesses. For example, loops can be scheduled with dynamic strategies and the resulting work distribution can be stored after the execution of the parallel loop in form of a template distribution. The same approach can be used for irregular codes to schedule computation onto the same processor when accessing data in the same coherence unit [BeGe 95a]. This *aligned-scheduling* is based on the performance monitoring support of the ASVM explained below.

The example in Figure 1 illustrates the usual usage of the work distribution features of SVM-Fortran. The parallel loop is scheduled via the distribution of template `T1`.

Besides the work distribution features of task parallel languages, several runtime mechanisms proved to be very useful, i.e. prefetching, poststore, page locking, and reduction handling.

Reduction operations are very important in scientific codes. All task parallel languages, KSR Fortran, Fortran-S, and SVM-Fortran provide an additional annotation to parallel loops to identify reduction variables. For example, in SVM-Fortran a parallel loop provides the option `reduction(var-name)` which defines the specified variable to be a reduction variable. The variable can be a scalar or an array and it is the task of the compiler to determine the reduction operation from the loop body. Instead of implementing a reduction variable as a shared variable using appropriate synchronization, each processor performs a local reduction on a private copy and the copies are merged into a global value at the end of the loop.

Page locking is also provided in all mentioned languages. In some codes it is necessary to implement atomic operations on data structures. This can be done via a critical section, but here, a portion of the code is defined to be atomic and thus only one processor can execute the code. This is too restrictive. The synchronization has to ensure that operations on the same data structure are atomic, but the same operation on other data structures can be executed in parallel, e.g. for different array elements. This is implemented by page locking which allows to lock all variables on a page.

SVM-Fortran as well as KSR Fortran provide prefetching of variables to hide memory access latency. While both languages allow to annotate individual references to a variable with a prefetch operation, SVM-Fortran also provides an interesting prefetch library, which monitors the page faults occuring in a parallel loop and stores the page faults in a *prefetch list*. Prefetch lists do have a unique index and a prefetch operation can be performed for a prefetch list everywhere in the program. This concept allows the programmer to provide additional overlap between prefetching and computation. An example for the usage of the SVM-Fortran prefetch library is shown in Figure 2. In the first incarnation of the parallel loop, the prefetch list with index `1` is constructed. During the subsequent executions of this code, the pages are prefetched and the `create_pfl` routines return immediatly since the prefetch list is already defined.

In addition to prefetching of variables also broadcasting a variable in advance to potential readers can help in optimizing applications with a single producer multiple consumer situation. This concept was introduced in KSR Fortran and called *poststore*. The potential readers can either be all processors or the subset of processors that had a read copy of the variable.

4 Performance Analysis

Programming SVM systems is not as simple as one could expect. Still the problem of access latency to remote adresses dominates program development. The user has to carefully tune his application to reduce

```
csvm$ replicated_region
100   call prefetch_pfl(1)
      ...
      call create_pfl_start(1,u,n*n*4)
csvm$ pdo(loops(j))
      do j=2,n-1
         do i=2,n-1
            uhelp(i,j)=..u(i-1,j)+u(i+1,j)
         enddo
      enddo
      call create_pfl_stop(1)
csvm$ replicated_region_end
```

Figure 2: SVM-Fortran prefetching library

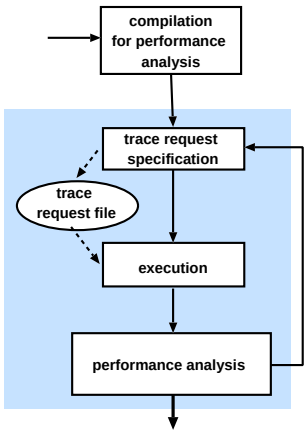

Figure 3: Incremental performance analysis

remote accesses resulting from true and false sharing. There are at least two important advantages compared to programming DM systems with HPF.

First, the program can be ported to the new architecture in an incremental approach. This approach is not applicable for HPF since the memory of a node severely limits the amount of data replication usually occuring for non-optimized code sections.

Second, during code optimization the user need not switch to another programming model. Optimizing HPF codes requires from the user a deep understanding of the generated message passing code while the users motivation for using HPF was to avoid message passing. Tools for optimizing programs for SVM systems can instead be implemented based on the source code of the application.

On SVM systems as well as on other systems standard profiling tools can be used to analyze the execution profile of an application. Usually, this information is not precise enough to understand the performance problems of an application.

In addition to a graphical profiling tool, KSR provided a trace-based performance analysis tool for visualizing start and stop events for program regions, such as parallel loops and parallel sections. Beyond this functionality, programming SVM systems requires a precise understanding of data movement and synchronization overhead. This precise knowledge cannot be obtained from statistics for the whole program printed at the end of a program run. A much finer granularity of information is required with a finer resolution in space as well as time, i.e. selectively gathered for code sections and different incarnations of these sections.

Gathering much finer information selectively for different incarnations is known as *tracing*. This technique is implemented in a number of performance analysis tools for the message passing model [Hond 95]. The tools trace information about message transfers and visualize the information graphically in form of graphical statistics and space-time diagrams. The two major deficiencies of these tools are the lack of source code information and the amount of data gathered at run time.

In the Fortran-S/Koan project, the trace-based performance visualization tool SVMview was developed based on the visualization widgets of the Pablo environment. The tool is based on a trace format which includes start and stop events for program regions and events for individual page faults. Based on the region events, the execution point in the source code can be visualized and based on the page fault information, different visualizations of the data movement are supported. This environment does not include features for reducing the amount of generated data.

In SVM systems tracing individual page faults leads to very big trace files which cannot be generated efficiently and also not be handled efficiently by the performance analysis tools.

In the SVM-Fortran project a modified approach is used to do performance analysis. The tool OPAL (Optimizer and Performance Analyzer) [GKO 95] allows to incrementally analyze the performance of a program. Instead of generating every information in a single run and selecting useful information afterwards in the analysis tool, very specific information can be requested from a program run. The incremental performance analysis concept is outlined in Figure 3.

The SVM-Fortran program is compiled once for performance analysis. The compiler inserts hooks which may call the performance analysis monitor SAM

(SVM-Fortran Application Monitor). Whether the hooks call the monitor is determined by a *trace request file*. In the trace request file the user specifies which information should be traced. Examples for such requests are:

request (∗) *local *.reg_no(*): RPFsum, WPFsum*
request (∗) *local foo.reg_(5): RPFsum(A,B)*
request (∗) *local foo.reg_no(5): RPF(A)*

The first request specifies that in all processors and all regions, i.e. parallel loops, parallel sections, and subroutines, the sums of the read and write page faults have to be traced. Thus, for each incarnation of the section the information including an identification of the section is written to a trace file. This demonstrates two features of the SVM-Fortran trace format: source code support and a hierarchy of trace information. The requested sums allow to determine regions with high page fault rates and, more precisely, the incarnations with high page fault rates.

The second request determines for a specific region in subroutine `foo` the individual page fault sums for specific arrays. This typically leads to the array responsible for a large number of page faults.

The last request generates individual events for each page fault. This precise information is used in the analysis of the program behaviour to determine exactly the array element access responsible for a page fault.

The requested information is generated either directly by SAM or SAM instruments the ASVM to trace information only available to the kernel. For example, region start and stop events are generated by SAM while page fault events are generated by the ASVM. To reduce intrusion, the events are collected in a trace buffer and flushed to files when the trace buffer is filled. SAM checks the fill rate of the buffer at every barrier and every region exit. Since the threshold for the fill rate at a barrier is lower than for a region, the buffer is typically flushed in barriers, disturbing all processes in the same way.

OPAL assists the user in all performance analysis phases on the basis of a source-code-related view. Program units are accessible through a graphical visualization of the call graph. The source code can be folded such that only introductory directives of parallel regions are visible, and comments can be hidden. These techniques allow the user to achieve a good overview of his program code.

OPAL supports a menu-driven specification of trace requests that frees the user from the subtle specification syntax. OPAL extracts the most useful statistical data, but does not store all the trace information in memory. These statistical data can be visualized as annotations to the source code in a separate performance column in the main window. This way of presenting information does provide a good overview of the performance results.

If the user needs more information on individual trace regions, he can mark a trace region and request a list of all trace events of a specific type. For example, OPAL presents all shared variables accessed in the selected trace region, and the user can select specific variables for which he would like to see individual page faults. The tool then extracts from the trace file of an individual processor only the pagefault information for these variables and the selected program region. Due to the source code information in the trace files, all the information can be requested and presented in form of trace regions and program variables, e.g. the faulting address is translated into the array name and the indices of the array element.

Thus, the tool allows to instrument selected regions of the source code to obtain certain information. Starting from statistical data for these program regions also incarnation-specific information can be analyzed for regions of the source code. Of course, sometimes time dependent information can better be visualized in a graphical manner. Therefore, the SVM-Fortran programming environment includes the state-of-the-art visualization tool PARvis (Parallel Visualization Environment). PARvis is mainly used for visualizing performance information of message passing codes. It also provides special displays supporting SVM-Fortran programs, e.g. an extension of the time line display with page movement information. Especially the time line display is extremly useful for obtaining a detailed understanding of the movement of individual pages. The required and provided interface technology is flexible zooming and scrolling of the time line which is implemented only in this tool.

The major drawback of such an environment is that still the user has to navigate through the large number of analysis choices and has to detect bottlenecks by himself. Future tools have to include user guidance as well as the ability to automatically analyze program performance. For frequently occuring performance bugs, the tool should be able to automatically proof the existence of these bugs by incrementally analyzing the program behaviour.

5 Program Optimization

Although the parallelization of applications is significantly simplified with the task parallel programming model on top of SVM systems, programs suffer from several performance bottlenecks. Performance

critical areas are data locality, load balance, and synchronization.

Reasons for imperfect data locality are true and false sharing. Required communication among processes due to data dependences is implemented in form of access to remote data. The resulting data migration is a consequence of true data sharing.

Reduction of true sharing can be implemented by elimination of data dependences or by localizing data dependences. In the literature a large number of loop transformations is known for this purpose. In addition to loop transformations, changing the work distribution to localize data dependences is an important technique. Automatic data and work distribution techniques play a central role in this context. In addition to static optimizations, run time optimization can be applied. Based on performance monitoring, the current page distribution can be determined and parallel iterations be executed on the processor owning the data due to previous computations. This aligned scheduling technique was first implemented in the SVM-Fortran project.

If communication cannot be avoided, prefetching and poststore are well known techniques to hide access latencies. As long as the access latency in SVM systems is very high, the prefetch operations have to be executed early before the access to allow enough overlapping of computation and communication. Prefetching can also eliminate double page faults, i.e. a write page fault following immediatly a read page fault for the same location.

Data migration also results from access to the same coherence unit without accessing the same data, i.e. false sharing. This problem is more important with larger coherence units, but it is well-known also in CC-RMA systems based on cache lines. Several techniques have been developed to reduce false sharing in programs. Again, most of the techniques optimize the work distribution, such as loop blocking and aligned scheduling. In addition to these techniques, transformations of the data mapping are applied, such as aligning arrays at page boundaries, array padding, dimension interchanging, and general linear transformations on the array shapes [AAL 95, BGM 95]. For irregular access patterns an element-wise reordering can be applied, such as renumbering techniques for finite element nodes [ToAb 94]. As mentioned at the beginning, also weak coherence protocols can be applied to reduce false sharing.

The second performance bottleneck is load balancing. Optimization of load balance should be done statically. If dynamic techniques, such as self-scheduling are applied, careful consideration of the impact on data locality is necessary. Therefore, more global techniques, such as redistribution of templates are preferable over loop-level techniques since a single page fault resulting from the migration of work will distort the comutation time due to high access latency.

A specific problem of the task parallel programming model is the amount of barrier synchronization. Barriers are executed before and after each parallel operation. At least languages have to provide annotations to eliminate barrier operations but it is preferable to perform barrier reduction automatically in the compiler. The techniques can either be based only on dependence information or can also take distribution information into account [BoBo 95, Tseng 95]. In the latter case, only dependences across processor shall not exist.

Besides elimination of barriers, barriers can also be replaced by pointwise synchronization. This is related very much to HPF compilation techniques, but has not been investigated in the context of task parallel languages. The problem of barrier overhead can of course be significantly reduced by faster barriers, as they are implemented in hardware on the Cray T3D system.

6 Summary

Shared virtual memory systems, either implemented in software or in hardware, are special distributed shared memory systems. The important characteristic is the migration of data among the physical memories, extending the concept of caching beyond the relatively small processor and secondary caches.

Programming languages based on the task parallel programming model, which is more general than the data parallel model, can be implemented in a straightforward way. Nevertheless, due to the physical distribution of memory, careful programming is necessary to obtain efficient codes. This programming has to be supported by adequate high-level programming tools for performance analysis and program optimization. Without these tools, which in our opinion are easier to build compared to similar tools for HPF, programming DSM systems is not much simpler than message passing programming. But, this is true for all high-level programming models trying to hide intricate details of parallel architectures.

References

[AAL 95] J.M. Anderson, S.P. Amarasinghe, M.S. Lam, *Data and Computation Transformations for Multiprocessors*, Proceedings of Fifth ACM SIG-

PLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95), Santa Barbara, pp. 134-143, 1995

[BGM 95] F. Bodin, E.D. Granston, T. Montaut, *Page-level Affinity Scheduling for Eliminating False Sharing*, Proceedings Fifth Workshop on Compilers for Parallel Computers, Malaga, pp. 175-186, 1995

[BKP 93] F. Bodin, L. Kervella, T. Priol, *Fortran-S: A Fortran Interface for Shared Virtual Memory Architectures*, Proceedings of Supercomputing 93, Portland, pp. 274-283, November 1993

[BeGe 95] R. Berrendorf, M. Gerndt, *SVM-Fortran Reference Manual Version 1.4*, Internal Report KFA-ZAM-IB-9510, Central Institute for Applied Mathematics, Research Centre Jülich, 1995

[BeGe 95a] R. Berrendorf, M. Gerndt, *Compiling Data Parallel Languages for Shared Virtual Memory Systems*, Internal Report KFA-ZAM-IB-9517, Central Institute for Applied Mathematics, Research Centre Jülich, 1995

[BoBo 95] M. O'Boyle, F. Bodin, *Compiler Reduction of Synchronization in Shared Virtual Memory Systems*, Proceedings ICS 95, Barcelona, pp. 318-327, 1995

[BoPr 92] F. Bodin, T. Priol, *Overview of the KOAN Programming Environment for the iPSC/2 and Performance Evaluation of the BECAUSE Test Program 2.5.1*, Proc. of BECAUSE European Workshop, Sophia-Antipolis, Oct. 1992

[GKO 95] M. Gerndt, A. Krumme, S. Özmen, *Performance Analysis for SVM-Fortran with OPAL*, submitted for publication to International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'95), Georgia, 1995

[HPF 94] HPFF, *High Performance Fortran Language Specification, Version 1.1, November 1994*, Rice University, Houston, Texas, November 1994

[Hond 95] A. Hondroudakis, *Performance Analysis Tools for Parallel Programs*, Edingburgh Parallel Computing Centre, The University of Edingburgh, 1995

[JLGS 90] D.V. James, A.T. Laundrie, S. Gjessing, G.S. Sohi, *Distributed-Directory Scheme: Scalable Coherent Interface*, IEEE Computer, Vol. 23, No. 6, pp. 74-77, 1990

[KSRF 92] Kendall Square Reserach Corporation, *KSR Fortran Programming Manual*, Kendall Square Research, 1992

[LLGW 92] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, M. Lam, *The Standford Dash Multiprocessor*, IEEE Computer, pp. 63-79, March 1992

[Li 86] K. Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors*, Ph.D. Dissertation, Yale University 1986, Technical Report YALEU/DCS/RR-492

[Li 88] K. Li, *IVY: A Shared Virtual Memory System for Parallel Computing*, ICPP 1988, Vol II, pp. 94-101, 1988

[SFL 94] I. Schoinas, B. Falsafi, A.R. Lebeck, S.K. Reinhardr, J.R. Larus, D.A. Wood, *Fine-grain Access Control for Distributed Shared Memory*, Proceedings ASPLOS-VI, ACM Operating Systems Review, 28(5), pp. 297-306, December 1994

[ToAb 94] K.A. Tomko, S.G. Abraham, *Data and Program Restructuring of Irregular Applications for Cache-Coherent Multiprocessors*, Proceedings ICS94, Manchester, pp. 214-225, 1994

[Tseng 95] C.-W. Tseng, *Compiler Optimizations for Eliminating Barrier Synchronization*, Proceedings PPOPP'95, ACM SIGPLAN Notices, Vol. 30, No. 8, pp. 144-155, August 1995

[ZTM 96] S. Zeisset, S. Tritscher, M. Mairandres, *A New Approach to Distributed Memory Management in the Mach Microkernel*, to appear in: Proceedings of the USENIX 1996 Technical Conference, San Diego, California, January 1996

[Zeis 93] S. Zeisset, *Evaluation and Enhancement of the Paragon Multiprocessor's Shared Virtual Memory System*, Diploma Thesis, Technische Universität München, 1993